

Сообщение о языке Active Oberon

Patrik Reali *

27 октября 2004 г.

1 Введение

Active Oberon является расширением оригинального языка Oberon [29, 30]. Его цель — введение в язык свойств для выражения параллелизма посредством *активных объектов (active objects)*. Данный отчет предполагает знакомство с языком Oberon; описываются только расширения языка.

Проектирование расширения языка направлялось на достижение унификации и гармонии. Изменения основаны на устоявшихся концепциях, таких как область действия и локальность. Обоснования, выходящие за рамки Active Oberon, описаны в [10].

1.1 Благодарности

Большое спасибо В. Kirk, А. Fischer, Т. Frei, J. Gutknecht, D. Lightfoot, и Р. Muller за рецензию данного документа, за внесение поправок и улучшений, а так же Владимиру Лосю за усовершенствование примера «барьер».

1.2 Предыстория и родственные работы

Разработка языков программирования в ETH Zurich имеет богатые традиции. Язык Oberon — это последний наследник семейства Algol, Pascal и Modula. Pascal [16] задумывался как язык для представления маленьких программ; простота и компактность сделали его особенно подходящим для обучения программированию. Modula [28] эволюционировала из Pascal как язык для системного программирования, и извлекла пользу из

*Перевод Андреева М. В.

практического опыта, полученного при проектировании рабочей станции Lilith [22] и операционной системы Medos [17]. Необходимость поддержки парадигмы «программирования в большом» была стимулом для создания Oberon [29]. Платформы Ceres [6] и Chameleon [12], операционная система Oberon [11] — эти проекты разрабатывались параллельно с проектированием языка, что позволило испытать и оценить его удобство.

Множество расширений языка Oberon было предложено как в ЕТН, так и вне его. Object Oberon [19], Oberon-2 [18], и Froderon [7] исследуют добавление дополнительных объектно-ориентированных свойств в язык; Oberon-V [9] предлагает дополнения для поддержки параллельных операций на векторных компьютерах; Oberon-XSC [15] добавляет математические возможности для поддержки научных вычислений; также было предложено встраивание модулей [24]. Параллелизм был впервые добавлен в операционную систему через специальный системный API в Concurrent Oberon [25] и XOberon [2]; попытка моделирования параллелизма средствами самого языка была предпринята Radenski [23].

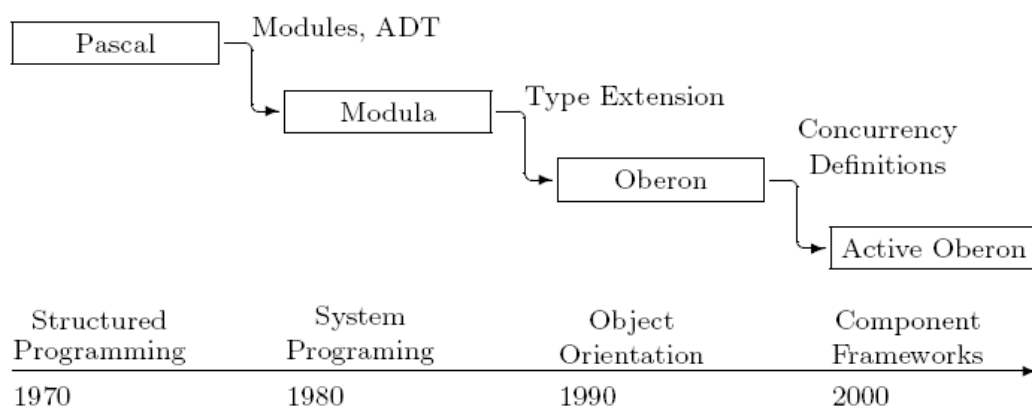


Рис. 1: Эволюция языков семейства Pascal

Active Oberon — это первый представитель нового поколения языков в данном семействе. Мы старались добавить в язык поддержку параллелизма и моделирования компонентов ясным и безболезненным способом.

1.3 Дизайн языка

На дизайн Active Oberon повлиял опыт, полученный при проектировании Oberon и Oberon-2. Мы следуем нотации Object Oberon для объявления классов и методов, т. к. считаем, что данный способ выразительнее нотации Oberon-2: методы принадлежат классу и, следовательно, должны

быть объявлены в классе; таким образом, прочие методы и поля, лежащие в области видимости записи, могут быть доступны без указания спецификатора. Защита от одновременного доступа при помощи модификатора `EXCLUSIVE` более читабельна, если методы принадлежат одной области видимости. Active Oberon отходит от дизайна Object Oberon в том, что записи одновременно являются так же и классами, т.е. не позволяет сосуществование классов и записей в одной системе. Другое важное отличие продиктовано решением позволить компилятору обрабатывать опережающие ссылки (`forward reference`). Синтаксис Object Oberon и Oberon-2 разработан в том числе и с целью упрощения создания компилятора, мы же постарались упростить работу программистов путем исключения ненужной избыточности опережающих объявлений, переложив тем самым работу на плечи компилятора.

Java [8] и C# [4] разделяют некоторые похожие идеи с Active Oberon. Они так же являются объектно-ориентированными языками из мира императивных языков, и механизм защиты экземпляров объектов от одновременного доступа так же связан с *мониторами*. С другой стороны, они делают акцент на объектно-ориентированности в такой экстремальной манере, что методы и поля класса трактуются как частный случай методов и полей экземпляра объекта, т.к. они лежат в пространстве имен класса. Более того, в Java нет полноценной поддержки для статического размещения структур: все структуры размещаются в динамической памяти, даже определенные пользователем массивы констант; поэтому, что бы получить приемлимую скорость исполнения, программы Java нуждаются в использовании сложной и затратной оптимизации при компиляции. Все языки из семейства Oberon рассматривают модули и классы как ортогональные понятия, каждое из них обладает своей областью действия; семантика модуля отличается от семантики класса так, как показано в [26] (для сравнения, В. Meyer защищает противоположное мнение [18]): модули группируют статические компоненты и соответствующие реализации, и предоставляют примитивы для развертывания и структурирования. На деле Java и C# вводят концепции, такие как пакеты (`packages`), пространства имен (`namespaces`) и сборки (`assemblies`), которые фактически являются модулями, но только под другим именем. Мы думаем, что все еще остаются веские причины для того, что бы статические структуры и модули были частью языка программирования.

Оператор `AWAIT` предложен и исследован Brinch Hansen [3], который показал его концептуальную простоту и элегантность, но в то же время думал, что его нельзя реализовать эффективно. Мы снова предлагаем этот оператор в Active Oberon с уверенностью, что это значительное усовершенствование по сравнению с сигналами и семафорами потому,

что он вносит унификацию и ясность; это становится особенно очевидно при программировании в объектно-ориентированном стиле, для которого сигналы и семафоры совершенно не подходят в виду их неструктурного использования, т.к. они могут быть добавлены в программы совершенно произвольным образом. В диссертации Питера Мюллера (Pieter Muller) [21] доказывається, что, при определенных ограничениях, оператор `AWAIT` может быть реализован эффективно. Язык Ada 95 [14] тоже использует конструкт, названный *барьеры* (*barriers*), который семантически весьма похож на `AWAIT` в Active Oberon, но только с детализацией на уровне процедур.

Concurrent Oberon был первой попыткой реализовать параллелизм в системе Oberon. Это было сделано через специальный API, определяющий тип `Thread` и функции для создания, остановки и возобновления исполнения. Защита была реализована при помощи одного глобального системного замка (`system lock`). Но не были предложены примитивы синхронизации. Эта модель так же слаба для Active Oberon, т.к. блокировки и синхронизация тесно связаны (когда выполняется синхронизация, блокировка снимается), и блокирующий механизм слишком грубый; один замок сделает мультипроцессорную систему (в которой множество процессов выполняется одновременно) бесполезной.

2 Объектно-ориентированные расширения

2.1 Указатель на безымянные типы записей

TYPE

(* пример указателя на безымянные типы записей *)

(* указатели на именованные типы записей *)

Tree = POINTER TO TreeDesc;

TreeDesc = RECORD key: INTEGER; l, r: Node END;

(* указатели на безымянные типы записей *)

Node = POINTER TO RECORD key: INTEGER; next: Node END;

DataNode = POINTER TO RECORD (Node) data: Data END;

DataTree = POINTER TO RECORD (Tree) data: Data END;

Типы `Node` и `DataNode` — *указатели на безымянные типы записей (pointers to anonymous record types)*; `Tree` — *указатель на именованный тип записей (pointer to named record type)*.

Экземпляры данных типов могут быть размещены только в динамической памяти (при помощи NEW), статические экземпляры не доступны; это вызвано тем, что тип записей безымянный и невозможно определить переменную этого типа.

Типы RECORD и POINTER TO RECORD могут быть базовыми типами для указателя на безымянный тип записей; тип записей не может расширить указатель на анонимную запись, таким образом выполняется свойство, разрешающее только динамические экземпляры.

2.2 Объектные типы

TYPE

```
(* объектные типы *)
```

```
DataObj = OBJECT VAR data: Data; l, r: DataObj END DataObj;
```

DataObj — это *объектный тип (object type)*.

Синтаксис типа OBJECT отличается от синтаксиса типа POINTER TO RECORD; он должен следовать правилу [DeclSeq] Body вместо правила FieldList. Это подразумевает, что процедуры могут быть объявлены внутри объектного типа. Мы называем их *методы (methods)* или *связанные с типом процедуры (type-bound procedures)*.

Только объектный тип может расширить другой объектный тип. Экземпляры объектных типов должны размещаться динамически при помощи NEW, подчиняясь правилу, продиктованному инициализаторами (Раздел 2.4).

2.3 Связанные с типом процедуры

TYPE

```
Coordinate = RECORD x, y: LONGINT END;
```

```
VisualObject = OBJECT  
  VAR next: VisualObject;
```

```
  PROCEDURE Draw*; (*нарисовать данный объект*)
```

```
  BEGIN HALT(99); (*заставить расширения переопределять этот метод*)
```

```
  END Draw;
```

```
END VisualObject;
```

```
Point = OBJECT (VisualObject)
```

```
  VAR pos: Coordinate;
```

```

    PROCEDURE Draw*; (*перекрываем метод Draw, объявленный в VisualObject*)
    BEGIN MyGraph.Dot(pos.x, pos.y)
    END Draw;
END Point;

Line = OBJECT (VisualObject)
    VAR pos1, pos2: Coordinate;

    PROCEDURE Draw*;
    BEGIN MyGraph.Line(pos1.x, pos1.y, pos2.x, pos2.y)
    END Draw;
END Line;

VAR
    objectRoot: VisualObject;

PROCEDURE DrawObjects*;
    VAR p: GraphObject;
BEGIN
    (* рисуем все объекты из списка *)
    p := objectRoot;
    WHILE p # NIL DO p.Draw; p := p.next END;
END DrawObjects;

```

Процедуры, объявленные внутри объекта, называются *связанные с типом процедуры (type-bound procedures)* или *методы (methods)*. Методы ассоциируются с экземпляром типа и оперируют им; внутри реализации метода, если другой метод виден по правилам видимости Oberon, то он доступен без указания квалификатора. Метод может перекрывать другой метод с таким же названием унаследованный от базового типа записей, но при этом он должен обладать такой же сигнатурой. Признак видимости является частью сигнатуры.

Замечание: Мы решили объявлять методы в области видимости объекта потому, что они принадлежат области видимости записи и могут быть доступны только через экземпляры записи. Это упрощает определение методов (нет получателя как в Oberon-2 [20]) и доступ к полям и методам следует хорошо известным правилам видимости Oberon. Мы боялись того, что циклические ссылки станут проблемой (например, два объектных типа взаимно ссылаются друг на друга), но решили, что концептуальная элегантность языка более важна. Решение проблемы в ослаблении правил видимости — допустить опережающие ссылки на символы, объявленные позднее в исходном тексте (раздел 4.1). Альтернативный способ заключается в расширении опережающих описаний до описания всего типа (как в Object Oberon [19]). Мы отвергли данное решение в виду внесения ненужной избыточности в код и вместо этого переложили решение проблемы на компилятор.

Дан экземпляр объекта o типа T со связанными процедурами P и Q , $o.P$ — это вызов метода. Внутри метода типа T другой метод типа T может быть вызван как Q . Метод P может вызвать метод, который он перекрывает, как $P\uparrow$. Это супервызов и он может быть выполнен только внутри метода.

2.4 Инициализаторы

Метод, помеченный символом $\&$, является *инициализатором объекта* (*object initializer*). Этот метод вызывается автоматически при создании экземпляра объекта. Объектный тип может иметь не более одного инициализатора. Если он есть, он всегда общедоступен и может быть вызван явно, как метод; если отсутствует, то наследуется инициализатор базового типа. Инициализатор может иметь сигнатуру отличающуюся от сигнатуры унаследованного инициализатора базового типа, но в этом случае наименование инициализатора так же должно отличаться.

Если объектный тип T содержит или наследует инициализатор P с сигнатурой $(p0: T0; \dots pn: Tn)$, тогда конкретизация переменной $o:T$ при помощи NEW требует указать параметры инициализатора: NEW(o , $p0$, \dots , pn). Инициализатор выполняется автоматически вместе с NEW.

Замечание: *Необязательный инициализатор делает возможным параметризацию типа. В частности, это весьма удобно при работе с активными объектами, т.к. параметризация экземпляра должна быть выполнена до старта активности. Если на чистоту, то нам такая нотация не нравится, но это единственный способ, который мы смогли придумать, добавить свойство не меняя сам язык.*

TYPE

```
Point = OBJECT (VisualObject)
  VAR pos: Coordinate;
```

```
  PROCEDURE & InitPoint(x, y: LONGINT);
  BEGIN pos.x := x; pos.y := y
  END InitPoint;
```

```
END Point;
```

```
PROCEDURE NewPoint(): Point;
```

```
  VAR p: Point;
  BEGIN NEW(p, x, y); (* вызывает NEW(p) и p.InitPoint(x, y) *)
  RETURN p
  END NewPoint;
```

2.5 SELF

Ключевое слово **SELF** может быть использовано в любом методе или в любой локальной процедуре метода объекта. Тип данной переменной совпадает с типом объекта и ее значение равно экземпляру объекта, к которому привязан метод. Переменная используется для доступа к объекту в случае, если нужна ссылка на объект или когда поле или метод объекта перекрывается другим символом, например, поле записи перекрывается локальной переменной с тем же именем.

TYPE

```
ListNode = OBJECT
  VAR data: Data; next: ListNode;
```

```
  PROCEDURE & InitNode (data: Data);
  BEGIN
```

```
    SELF.data := data; (* инициализируем данные объекта *)
```

```

        next := root; root := SELF (* добавляем node в начало списка *)
    END InitNode;
END ListNode;

```

```

VAR
    root: ListNode;

```

2.6 Делегаты

```

TYPE
    MediaPlayer = OBJECT
        PROCEDURE Play; .... показать фильм .... END Play;
        PROCEDURE Stop; .... остановить фильм .... END Stop;
    END MediaPlayer;

    ClickProc = PROCEDURE {DELEGATE};

    Button = OBJECT
        VAR
            onClick: ClickProc;
            caption: ARRAY 32 OF CHAR;

            PROCEDURE OnClick;
            BEGIN onClick END OnClick;

            PROCEDURE & Init(caption: ARRAY OF CHAR; onClick: ClickProc);
            BEGIN SELF.onClick := onClick; COPY(caption, SELF.caption)
            END Init;
        END Button;

    PROCEDURE Init(p: MediaPlayer);
        VAR b0, b1, b2: Button;
    BEGIN
        (* Перезагрузка -> вызвать системную функцию *)
        NEW(b0, "Reboot", System.Reboot);

        (* Интерфейс MediaPlayer: связываем кнопки с экземпляром плеера *)
        NEW(b1, "Play", p.Play);
        NEW(b2, "Stop", p.Stop);
    END Init;

```

Делегаты подобны процедурным типам; они совместимы как с процедурами так и с методами, в то время как процедурные типы совместимы только с процедурами.

Делегаты процедурных типов помечаются модификатором `DELEGATE`. Делегатам могут быть назначены процедуры и методы. Пусть даны переменная с процедурным типом t , экземпляр объекта o и метод M связанный с o , тогда можно записать $o.M$ в t , если конечно метод и t обладают совместимыми сигнатурами. Ссылка на сам объект исключается из сигнатуры процедурного типа. Всякий раз при вызове t назначенный объект o явно передается в self-ссылку (self-reference). Присваивания и вызовы процедур остаются совместимыми с описанием Oberon.

2.7 Описания (definitions)

Описание (definition) — это синтаксический контракт ¹, определяющий набор сигнатур методов. Описание $D0$ может быть *уточнено* новым описанием $D1$, которое наследует все методы, объявленные в $D0$. Описания и их методы видимы глобально. Объект может реализовать одно или несколько описаний, в этом случае он обязуется реализовать все методы определенные в этих описаниях.

```
DEFINITION Runnable;  
  PROCEDURE Start;  
  PROCEDURE Stop;  
END Runnable;
```

```
DEFINITION Preemptable REFINES Runnable;  
  PROCEDURE Resume;  
  PROCEDURE Suspend;  
END Preemptable;
```

```
TYPE  
  MyThread = OBJECT IMPLEMENTS Runnable;  
    PROCEDURE Start;  
    BEGIN .... END Start;  
  
    PROCEDURE Stop;  
    BEGIN .... END Stop;
```

¹[1] описывает четыре уровня контрактов: 1. синтаксический контракт (система типов); 2. поведенческий контракт (инварианты, предусловия и постусловия); 3. контракты синхронизации; 4. контракты качества услуг

```
END MyThread;
```

Ключевое слово `IMPLEMENTS` используется для указания описаний, реализованных объектным типом. Объектный тип может реализовать несколько описаний.

Описания можно понимать как дополнительные свойства, которыми должен обладать объект, но которые ортогональны иерархии типов объектов. Метод объекта может быть вызван через описание, в этом случае во время исполнения проверяется, действительно ли экземпляр объекта реализует описание, и только после этого вызывается метод; если экземпляр объекта не реализует описание, то возникает исключение (`run-time exception`).

```
PROCEDURE Execute(o: OBJECT; timeout: LONGINT);
BEGIN
  Runnable(o).Start;
  Delay(timeout);
  Runnable(o).Stop;
END Execute;
```

3 Поддержка параллелизма

3.1 Активные объекты

Определение объекта может включать `StatBlock`, названное *телом объекта* (*object body*). Тело — это активность объекта, которая выполняется после того, как экземпляр объекта размещен и инициализатор (если он есть) завершил свое исполнение; тело объекта помечается модификатором `ACTIVE`. Во время размещения объекта так же размещается новый процесс, который исполняет тело параллельно; такой объект называется *активным объектом* (*active object*).

Если не указан модификатор `ACTIVE`, тело выполняется синхронно; выход из `NEW` происходит только после того, как исполнение тела объекта завершается.

Система сохраняет явные ссылки на активные объекты до завершения исполнения активности для того, чтобы избежать утилизацию объекта в процессе сборки мусора. Объект может пережить свою активность.

```
TYPE
  (* определяем объект и его поведение *)
  Object = OBJECT
```

```

BEGIN {ACTIVE} (* тело объекта *)
  ... делаем что-либо ...
END Object;

```

```

PROCEDURE CreateAndStartObject;
  VAR o: Object;
BEGIN
  ... NEW(o); ...
END CreateAndStartObject;

```

Активность объекта завершается после завершения исполнения тела объекта. Пока исполняется тело, объект продолжает существовать (например, он не может быть утилизирован при сборе мусора). После этого объект становится пассивным и может быть утилизирован в соответствии с обычными правилами.

3.2 Защита

Statement Block — это последовательность операторов, заключенная между `BEGIN` и `END`. Он может использоваться в любом месте как и простой оператор. Более полезно использовать его вместе с модификатором `EXCLUSIVE` для создания критической области для защиты операторов от одновременного исполнения.

```

PROCEDURE P;
  VAR x, y, z: LONGINT;
BEGIN
  x := 0;
  BEGIN
    y := 1
  END;
  z := 3
END P;

```

Объект может рассматриваться как ресурс и различные активности могут потенциально соревноваться за его использование или за эксклюзивный доступ к предоставляемым им средствам; в таком случае защита доступа необходима. Наша модель защиты — монитор, размещенный в экземпляре объекта (*instance-based monitor*).

```

(* Процедуры Set и Reset взаимно исключаемы *)
TYPE

```

```

MyContainer = OBJECT
  VAR x, y: LONGINT; (* Инвариант: y = f(x) *)

  PROCEDURE Set(x: LONGINT);
  BEGIN {EXCLUSIVE} (* изменение x и y атомарно *)
    SELF.x := x; y := f(x)
  END Set;

  PROCEDURE Reset;
  BEGIN
    ...
    BEGIN {EXCLUSIVE} (* изменение x и y атомарно *)
      x := x0; y := y0;
    END;
    ....
  END Reset;
END MyContainer

```

Каждый экземпляр объекта защищен и единицей защиты является произвольный блок операторов от отдельного оператора до целого метода. Блок операторов может быть защищен от одновременного доступа при помощи модификатора **EXCLUSIVE**. Активность остается на входе в эксклюзивный блок до тех пор, пока другая активность находится в эксклюзивном блоке того же экземпляра объекта.

Активность не может заблокировать объект более одного раза, повторный вход не допускается.

Каждый модуль считается объектным типом с *единственным экземпляром (singleton instance)*, таким образом его процедуры тоже могут быть защищены. Областью видимости защиты является модуль целиком как и в случае мониторов [13].

Замечание: *Реализована только EXCLUSIVE блокировка: SHARED блокировки (как это описано в [10]) могут быть реализованы через EXCLUSIVE блокировки и соответственно не являются базовой концепцией. Они используются очень редко и их реализация не оправдывает усложнение языка. Реализация SHARED блокировок описана в Приложении В.1.*

Замечание: Повторный вход в блокировку не поддерживается из-за концептуальной нечистоты (см. [27]) и его корректная обработка стоит дорого; в нем нет реальной необходимости, т.к. можно проектировать программы без его использования. Повторно-входимые блокировки могут быть реализованы при помощи простых блокировок (см. Приложение В.3).

3.3 Синхронизация

TYPE

```
Synchronizer = OBJECT
  awake: BOOLEAN

  PROCEDURE Wait;
  BEGIN {EXCLUSIVE} AWAIT(awake); awake := FALSE
  END Wait;

  PROCEDURE WakeUp;
  BEGIN {EXCLUSIVE} awake := TRUE
  END WakeUp;
END Synchronizer;
```

Встроенная процедура `AWAIT` используется для синхронизации активности с состоянием системы. Аргументом `AWAIT` может быть только логическое *условие*; активность сможет продолжить свое выполнение только после того, как *условие* станет истинным. Пока условие не выполняется, активность остается *приостановленной* (*suspended*); если это происходит внутри защищенного блока, то блокировка снимается на время приостановки активности (что позволяет другим активностям изменять состояние объекта и сделать условие истинным); активность возобновляет работу только если она сможет снова захватить блокировку.

Система отвечает за проверку условий и за возобновление работы приостановленных активностей. Условия внутри экземпляра объекта перепроверяются в случае, когда некоторая активность выходит из защищенного блока того же самого экземпляра объекта. Это означает, что изменение состояния объекта вне защищенного блока не приводит к переычислению условия.

Когда несколько активностей соревнуются за одну и ту же блокировку, то активности с выполненными условиями рассматриваются раньше тех, которые только хотят войти в защищенную область.

Приложение В.6 демонстрирует синхронизацию внутри разделяемого буфера.

Замечание: *Синхронизация зависит от состояния объекта, например, ожидание доступности некоторых данных или состояния для изменения. Объект используется как контейнер для данных и любой доступ осуществляется через защищенные методы или блоки. Мы подразумеваем, что при каждом доступе к защищенному блоку происходит изменение состояния объекта; таким образом условия перепроверяются только в этот момент. Это означает, что изменение состояния объекта вне защищенного блока не приводит к превычению условия. Для принудительной проверки условий можно вызвать пустой защищенный метод или войти в пустой защищенный блок.*

4 Прочие расширения языка

В данном разделе описываются несколько важных изменений, сделанных для лучшего интегрирования расширений в язык.

4.1 Последовательность определений и опережающие ссылки

В Active Oberon область видимости описания символа распространяется на весь блок, содержащий его. Это означает, что символ может быть использован до своего определения, и что имена уникальны внутри области видимости.

4.2 HUGEINT

В язык был добавлен 64 битный знаковый целый тип HUGEINT. Он вписывается в иерархию числовых типов следующим образом:

$$\text{LONGREAL} \supseteq \text{REAL} \supseteq \text{HUGEINT} \supseteq \text{LONGINT} \supseteq \text{INTEGER} \supseteq \text{SHORTINT}$$

Таблица 1 показывает новые процедуры для изменения типа. Никаких новых правил описания констант не вводится; константы типизируются в соответствии с их значением.

<i>Имя</i>	<i>Тип аргумента</i>	<i>Тип результата</i>	<i>Функция</i>
SHORT(<i>x</i>)	HUGEINT	LONGINT	идентичность (возможно усечение)
LONG(<i>x</i>)	LONGINT	HUGEINT	идентичность
ENTIERH(<i>x</i>)	вещественный	HUGEINT	наибольшее целое, не превышающее <i>x</i>

Таблица 1: Новые процедуры изменения типа

4.3 Нетрассируемые указатели (untraced pointers)

Нетрассируемые указатели — это указатели, которые не отслеживаются сборщиком мусора. Структура или объект, на которые ссылаются только нетрассируемые указатели, могут быть в любой момент утилизированы сборщиком мусора.

Нетрассируемые указатели определяются при помощи модификатора UNTRACED.

```
TYPE Untraced = POINTER {UNTRACED} TO T;
```

4.4 Новое для IA32

Функции из таблицы 2 были добавлены в компилятор для платформы Intel IA32.

PUTx и GETx были добавлены ради безопасности, для работы с нетипизированными константами.

4.5 Прочее

Некоторые расширения из Oberon-2 были адаптированы для Active Oberon:

- ASSERT
- FOR
- экспорт только для чтения
- динамические массивы

Переменные указатели автоматически инициализируются значением NIL.

<i>Имя</i>	<i>Функция</i>
PUT8(adr: LONGINT; x: SHORTINT) PUT16(adr: LONGINT; x: INTEGER) PUT32(adr: LONGINT; x: LONGINT) PUT64(adr: LONGINT; x: HUGEINT)	Mem[adr] := x
GET8(adr: LONGINT): SHORTINT GET16(adr: LONGINT): INTEGER GET32(adr: LONGINT): LONGINT GET64(adr: LONGINT): HUGEINT	RETURN Mem[adr]
PORTIN(port: LONGINT; x: AnyType) PORTOUT(port: LONGINT; x: AnyType)	x := IOPort(port) IOPort(port) := x
CLI STI	отключить прерывания включить прерывания
EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP AX, BX, CX, DX, SI, DI AL, AH, BL, BH, CL, CH, DL, DH	PUTREG/GETREG константы 32-битовые регистры 16-битовые регистры 8-регистры

Таблица 2: Новое в модуле SYSTEM для IA32

Список литературы

- [1] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.
- [2] R. Brega. Real-time kernel for the Power-PC architecture. Master's thesis, Institut für Robotik, ETH Zürich, 1995.
- [3] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972. Reprinted in *The Search for Simplicity*, IEEE Computer Society Press, 1996.
- [4] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001.
- [5] Edsger W. Dijkstra. The structure of the THE-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.
- [6] H. Eberle. *Development and Analysis of a Workstation Computer*. Dissertation 8431, ETH Zürich, 1987.

- [7] P. Fröhlich. Projekt Froderon: Zur weiteren Entwicklung der Programmiersprache Oberon-2. Master's thesis, Fachhochschule München, 1997.
- [8] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. The Java Series. Addison-Wesley, 1st edition, 1996.
- [9] R. Griesemer. A Programming Language for Vector Computers. Dissertation 10277, ETH Zürich, 1993.
- [10] J. Gutknecht. Do the fish really need remote control? A proposal for selfactive objects in Oberon. In Proc. of Joint Modular Languages Conference (JMLC). LNCS 1024, Linz, Austria, March 1997. Springer Verlag.
- [11] J. Gutknecht and N. Wirth. Project Oberon - The Design of an Operating System and Compiler. Addison-Wesley, 1992.
- [12] B. Heeb and C. Pfister. Chameleon: A workstation of a different colour. In Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping. Second International Workshop on Field Programmable Logic and Applications, pages 152–161, August 1992.
- [13] C. A. R. Hoare. Monitors: An operating system structuring concept. Communications of the ACM, 17(10):549–557, October 1974. Erratum in Communications of the ACM, Vol. 18, No. 2 (February), p. 95, 1975. This paper contains one of the first solutions to the Dining Philosophers problem.
- [14] International Organization for Standardization. ISO/IEC 8652:1995: Information technology — Programming languages — Ada. International Organization for Standardization, Geneva, Switzerland, 1995.
- [15] P. Januschke. Oberon-XSC - Eine Programmiersprache und Arithmetikbibliothek für das Wissenschaftliche Rechnen. PhD thesis, Universität Karlsruhe, 1998.
- [16] K. Jensen and N. Wirth. PASCAL - User Manual and Report, volume 18 of Lecture Notes in Computer Science. Springer, 1974.
- [17] S. E. Knudsen. Medos-2: A Modula-2 oriented operating system for the personal computer Lilith. Diss no. 7346, ETH Zürich, 1983.
- [18] B. Meyer. Object-Oriented Software Construction. Prentice Hall, 2nd edition, 1997.

- [19] H. Mössenböck, J. Templ, and R. Griesemer. Object Oberon: An objectoriented extension of Oberon. Technical Report 1989TR-109, Department of Computer Science, ETH Zürich, June 1989.
- [20] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.
- [21] P.J. Muller. The Active Object System – Design and Multiprocessor Implementation. PhD thesis, ETH Zürich, 2002.
- [22] R. Ohran. Lilith: A Workstation Computer for Modula-2. Dissertation 7646, ETH Zürich, 1984.
- [23] A. Radenski. Introducing objects and concurrency to an imperative programming language. *Information Sciences, an International Journal*, 87(1- 3):107–122, 1995.
- [24] A. Radenski. Module embedding. *Software - Concepts and Tools*, 19(3):122–129, 1998.
- [25] B. A. Sanders and S. Lalis. Adding concurrency to the Oberon system. In *Proceedings of Programming Languages and System Architectures, Lecture Notes in Computer Science (LNCS) 782*. Springer Verlag, March 1994.
- [26] C. Szyperski. Import is not inheritance – why we need both: modules and classes. In O. Lehrmann Madsen, editor, *Proceedings, ECOOP 92*, number 615 in *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, 1992.
- [27] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.
- [28] N. Wirth. MODULA : A language for modular multiprogramming. *Software Practice and Experience*, 7:3–35, 1977.
- [29] N. Wirth. The programming language Oberon. *Software Practice and Experience*, 18(7):671–690, July 1988.
- [30] N. Wirth and M. Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.

A Синтаксис Active Oberon

```
Module      = MODULE ident ';' [ImportList] {Definition} {DeclSeq} Body ident '.'.
ImportList = IMPORT ident [':=' ident] {',' ident [':=' ident ]} ';'.
Definition = DEFINITION ident [REFINES Qualident] {PROCEDURE ident [FormalPars] ';' } END ident.
DeclSeq    = CONST {ConstDecl ';' } | TYPE {TypeDecl ';' } | VAR {VarDecl ';' } | {ProcDecl ';' }.
ConstDecl  = IdentDef '=' ConstExpr.
TypeDecl   = IdentDef '=' Type.
VarDecl    = IdentList ';' Type.
ProcDecl   = PROCEDURE ProcHead ';' {DeclSeq} Body ident.
ProcHead   = [SysFlag] ['*' | '&'] IdentDef [FormalPars].
SysFlag    = '[' ident ']'.
FormalPars = '(' [FPSection {';' FPSection}] ')' [':' Qualident].
FPSection  = [VAR] ident {',' ident} ':' Type.
Type       = Qualident
            | ARRAY [SysFlag] [ConstExpr {',' ConstExpr}] OF Type
            | RECORD [SysFlag] ['(' Qualident ')'] [FieldList] END
            | POINTER [SysFlag] TO Type
            | OBJECT [[SysFlag] ['(' Qualident ')']] [IMPLEMENTS Qualident] {DeclSeq} Body]
            | PROCEDURE [SysFlag] [FormalPars].
FieldDecl  = [IdentList ';' Type].
FieldList  = FieldDecl {';' FieldDecl}.
Body       = StatBlock | END.
StatBlock  = BEGIN ['{' IdentList '}'] [StatSeq] END.
StatSeq    = Statement {';' Statement}.
Statement  = [Designator ':' Expr
            | Designator ['(' ExprList ')']
            | IF Expr THEN StatSeq {ELSIF Expr THEN StatSeq}[ELSE StatSeq] END
            | CASE Expr DO Case {'|'} Case} [ELSE StatSeq] END
            | WHILE Expr DO StatSeq END
            | REPEAT StatSeq UNTIL Expr
            | FOR ident ':' Expr TO Expr [BY ConstExpr] DO StatSeq END
            | LOOP StatSeq END
            | WITH Qualident ':' Qualident DO StatSeq END
            | EXIT
            | RETURN [Expr]
            | AWAIT '(' Expr ')',
            | StatBlock
            ].
Case       = [CaseLabels {',' CaseLabels} ':' StatSeq].
CaseLabels = ConstExpr ['..' ConstExpr].
ConstExpr  = Expr.
Expr       = SimpleExpr [Relation SimpleExpr].
SimpleExpr = Term {MulOp Term}.
Term       = ['+' | '-' ] Factor {AddOp Factor}.
Factor     = Designator ['(' ExprList ')'] | number | character | string
            | NIL | Set | '(' Expr ')' | 'Factor'.
Set        = '{' [Element {',' Element}] '}'.
Element    = Expr ['..' Expr].
Relation   = '=' | '#' | '<' | '<=' | '>' | '>=' | IN | IS.
MulOp      = '*' | DIV | MOD | '/' | '&'.
AddOp      = '+' | '-' | OR.
Designator = Qualident {'.' ident | '[' ExprList ']' | '^'
            | '(' Qualident ')' }.
ExprList   = Expr {',' Expr}.
IdentList  = IdentDef {',' IdentDef}.
Qualident  = [ident '.' ] ident.
IdentDef   = ident ['*' | '-'].
```

В Примеры синхронизации

В.1 Читатели и писатели

```
MODULE ReaderWriter;  
  
TYPE  
  RW = OBJECT  
    (* n = 0, пусто *)  
    (* n < 0, n писателей *)  
    (* n > 0, n читателей *)  
    VAR n: LONGINT;  
  
    PROCEDURE EnterReader*;  
    BEGIN {EXCLUSIVE}  
      AWAIT(n >= 0); INC(n)  
    END EnterReader;  
  
    PROCEDURE ExitReader*;  
    BEGIN {EXCLUSIVE}  
      DEC(n)  
    END ExitReader;  
  
    PROCEDURE EnterWriter*;  
    BEGIN {EXCLUSIVE}  
      AWAIT(n = 0); DEC(n)  
    END EnterWriter;  
  
    PROCEDURE ExitWriter*;  
    BEGIN {EXCLUSIVE}  
      INC(n)  
    END ExitWriter;  
  
    PROCEDURE & Init;  
    BEGIN n := 0  
    END Init;  
  
  END RW;  
  
END ReaderWriter.
```

Образец Читатели – Писатели регулирует доступ к данным в критической секции. Либо один Писатель (активность, изменяющая состояние объекта), либо несколько Читателей (активности, не изменяющие состояние объекта) допускаются в критическую секцию в предоставленное время.

В.2 Сигналы

```
TYPE
Signal* = OBJECT
  VAR
    in: LONGINT; (* следующий билет для выдачи *)
    out: LONGINT; (* следующий билет для обслуживания *)

  (* элементы с (out <= ticket < in) должны ждать *)
  PROCEDURE Wait*;
    VAR ticket: LONGINT;
  BEGIN {EXCLUSIVE}
    ticket := in; INC(in); AWAIT(ticket - out < 0)
  END Wait;

  PROCEDURE Notify*;
  BEGIN {EXCLUSIVE}
    IF out # in THEN INC(out) END
  END Notify;

  PROCEDURE NotifyAll*;
  BEGIN {EXCLUSIVE}
    out := in
  END NotifyAll;

  PROCEDURE & Init;
  BEGIN in := 0; out := 0
  END Init;
END Signal;
```

`Signal` реализует примитивы для работы с сигналами Active Oberon подобно тому, как это сделано в Java и Modula-2. Он использует слегка измененный `ticket-algorithm`. Как в некоторых магазинах, каждый покупатель получает занумерованный билет, это гарантирует, что покупатели будут обслужены в порядке их прибытия.

В.3 Повторно входимые блокировки

```
ReentrantLock* = OBJECT
VAR
  lockedBy: PTR;
  depth: LONGINT;

PROCEDURE Lock*;
  VAR me: PTR;
BEGIN {EXCLUSIVE}
  me := AosActive.CurrentThread();
  AWAIT((lockedBy = NIL) OR (lockedBy = me));
  lockedBy := me;
  INC(depth)
END Lock;

PROCEDURE Unlock*;
BEGIN {EXCLUSIVE}
  DEC(depth);
  IF depth = 0 THEN lockedBy := NIL END
END Unlock;
END ReentrantLock;
```

`ReentrantLock` позволяет блокировать объект его хозяином более одного раза. Клиенты этого объекта должны явно использовать `Lock` и `Unlock` вместо пометки защищаемого участка оператором `EXCLUSIVE`.

В.4 Бинарный и общий семафоры

```
MODULE Semaphores;

TYPE
  Sem* = OBJECT (* Бинарный семафор *)
    VAR taken: BOOLEAN

    PROCEDURE P*; (* войти *)
    BEGIN {EXCLUSIVE}
      AWAIT(~taken); taken := TRUE
    END P;

    PROCEDURE V*; (* войти *)
    BEGIN {EXCLUSIVE}
      taken := FALSE
    END V;

    PROCEDURE & Init;
    BEGIN taken := FALSE
    END Init;
  END Sem;

  GSem* = OBJECT (* Общий семафор *)
    VAR slots: LONGINT;

    PROCEDURE P*;
    BEGIN {EXCLUSIVE}
      AWAIT(slots > 0); DEC(slots)
    END P;

    PROCEDURE V*;
    BEGIN {EXCLUSIVE}
      INC(slots)
    END V;

    PROCEDURE & Init(n: LONGINT);
    BEGIN slots := n
    END Init;
  END GSem;
END Semaphores.
```

Это хорошо известные синхронизирующие примитивы Дейкстры [5]. Заметим, что возможность реализовать семафоры показывает, что модель Active Oberon достаточно мощная для поддержки защиты и синхронизации параллельных процессов.

В.5 Барьеры

```
MODULE Barriers;
(*
  Барьер используется для синхронизации N активностей.
*)
TYPE
  Barrier = OBJECT
    VAR in, out, N: LONGINT;

    PROCEDURE Enter*;
      VAR i: LONGINT;
    BEGIN {EXCLUSIVE}
      INC(in);
      AWAIT (in >= N);
      INC(out);
      IF (out = N) THEN in := 0; out := 0 END;
    END Enter;

    PROCEDURE & Init (nofProcs: LONGINT);
    BEGIN
      N := nofProcs; in := 0; out := 0;
    END Init;
  END Barrier;
END Barriers.
```

Барьер используется для синхронизации активностей друг с другом. Если активности определены как

$$P_i = Phase_{i,0}; Phase_{i,1}; \dots Phase_{i,n}$$

то барьер используется для гарантии того, что все активности выполнят $Phase_{i,j}$ до начала $Phase_{i,j+1}$. Отдельный поток исполнения будет выглядеть подобно следующему:

```
FOR j := 0 TO N DO
  Phase(i, j); barrier.Enter
END;
```

Барьер сбрасывает счетчик *in* после выполнения условия чтобы избежать переполнения. Это возможно потому, что активности, повторно запрашивающие блокировку через инструкцию `AWAIT`, имеют более высокий приоритет по сравнению с активностями, запрашивающими блокировку в первый раз в том же блоке `EXCLUSIVE`.

В.6 Ограниченный буфер

```
MODULE Buffers;

CONST
  BufLen = 256;

TYPE
  (* Buffer - FIFO буфер *)
  Buffer* = OBJECT
    VAR
      data: ARRAY BufLen OF INTEGER;
      in, out: LONGINT;

      (* Put - вставить элемент в буфер *)
      PROCEDURE Put* (i: INTEGER);
      BEGIN {EXCLUSIVE}
        AWAIT ((in + 1) MOD BufLen # out); (*AWAIT ~полный *)
        data[in] := i;
        in := (in + 1) MOD BufLen
      END Put;

      (* Get - забрать элемент из буфера *)
      PROCEDURE Get* (VAR i: INTEGER);
      BEGIN {EXCLUSIVE}
        AWAIT (in # out); (*AWAIT ~пустой *)
        i := data[out];
        out := (out + 1) MOD BufLen
      END Get;

      PROCEDURE & Init;
      BEGIN
        in := 0; out := 0;
      END Init;

    END Buffer;

END Buffers.
```

`Buffer` реализует ограниченный кольцевой буфер. Методы `Put` и `Get` защищены от одновременного доступа; они так же проверяют наличие свободного места и данных соответственно, в противном случае активность приостанавливается до того, как место освободиться или поступят новые данные.

С Примеры ативных объектов

С.1 Обедающие философы

```
MODULE Philo;

IMPORT Semaphores;

CONST
  NofPhilo = 5; (* количество философов *)

VAR
  fork: ARRAY NofPhilo OF Semaphores.Sem;
  i: LONGINT;

TYPE
  Philosopher = OBJECT
    VAR
      first, second: LONGINT;

    (* вилки для философов *)
    PROCEDURE & Init(id: LONGINT);
    BEGIN
      IF id # NofPhilo-1 THEN
        first := id; second := (id+1)
      ELSE
        first := 0; second := NofPhilo-1
      END
    END Init;

  BEGIN {ACTIVE}
    LOOP
      ... Думает...
      fork[first].P; fork[second].P;
      ... Ест ...
      fork[first].V; fork[second].V
    END
  END Philosopher;

VAR
  philo: ARRAY NofPhilo OF Philosopher;
BEGIN
  FOR i := 0 TO NofPhilo DO
    NEW(fork[i]);
    NEW(philo[i]);
  END;
END Philo.
```

C.2 Решето Эратосфена

```
MODULE Eratosthenes; (* prk 13.09.00 *)

IMPORT Out, Buffers;

CONST
  N = 2000;
  Terminate = -1; (* охранник *)

TYPE
  Sieve = OBJECT (Buffers.Buffer)
    VAR prime, n: INTEGER; next: Sieve;

    PROCEDURE & Init;
    BEGIN
      Init^; (* вызывает инициализатор Buffer (суперклас) *)
      prime := 0; next := NIL
    END Init;

  BEGIN {ACTIVE}
    LOOP
      Get(n);

      IF n = Terminate THEN
        (* прервать выполнение *)
        IF next # NIL THEN next.Put (n) END;
        EXIT
      ELSIF prime = 0 THEN
        (* первое число всегда простое *)
        Out.Int(n, 0); Out.String(" простое"); Out.Ln;
        prime := n;
        NEW (next)
      ELSIF (n MOD prime) # 0 THEN
        (* передать дальше, если это не множитель простого *)
        next.Put (n)
      END
    END
  END Sieve;

  PROCEDURE Start*;
  VAR s: Sieve; i: INTEGER;
  BEGIN
    NEW(s);
    FOR i := 2 TO N-1 DO s.Put (i) END;
    s.Put(Terminate) (* использовать охранника для индикации выполнения *)
  END Start;

END Eratosthenes.
```

`Eratosthenes` использует отсеивающий алгоритм для поиска простых чисел. Каждое решето — это активный объект, который передает все полученные значения, не являющиеся множителем первого полученного числа, на следующее решето. Синхронизация осуществляется в буфере.

D Журнал версий

15 марта 2003	Улучшен пример «Барьеры»
4 апреля 2002	Незначительные изменения
14 марта 2003	Синтаксис определений; несколько небольших изменений
11 января 2002	Небольшие исправления, Statement Block перемещен в Protection
20 декабря 2001	История
9 августа 2001	Небольшие исправления, добавлен модификатор Delegate
19 апреля 2001	Объектные типы вместо динамических записей, множество небольших улучшений
25 марта 2001	Ревизия
20 февраля 2001	Добавлены делегаты